THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 60, No. 8, October 1981
Printed in U.S.A.

Using Documentation as a Software Design Medium

By S. D. HESTER, D. L. PARNAS,* and D. F. UTTER

(Manuscript received March 25, 1981)

This article describes a software design method based on the principles of separation of concerns and information hiding. The principle of separation of concerns is used to structure the design documentation, and information hiding is used to guide the internal design of the software. Separation of concerns requires that design information be divided into clearly distinct and relatively independent documents. The design documents are the main products of the initial design phase, and are carefully structured to (i) expose open issues, (ii) express design decisions, and (iii) ensure that information is recorded in a way that allows it to be readily retrieved later. Information hiding is used to design software that is easy to change. We have applied many elements of the design method to the development of the No. 2 Service Evaluation System (SES), a multiprocessor data acquisition and transaction system. Our experiences in applying the design method are described, and some examples are included.

I. INTRODUCTION

This article describes a software design method based on the principles of separation of concerns and information hiding. Software design documentation is the medium used to apply the principles.

The expected benefits of the design method are as follows:

(*i*) *Ease of change.* System functions that are likely to change are identified and information hiding is applied to minimize the amount of software affected by a change in these functions.

(*ii*) *Control of the information about the functions of the system.* A carefully structured requirements document is to be maintained throughout the life of the project.

* Currently on leave from University of North Carolina. Present addresses: IBM Federal Systems Division and Naval Research Laboratory.

(*iii*) *Ordering of the development steps to meet the project objectives.* Documentation of the useful subsets of the system and the dependencies between the software modules serve to guide the scheduling of the development effort.

(*iv*) *Making the agreements between developers explicit.* Misunderstandings are avoided and a smoother system integration is achieved by documenting the interfaces between the software modules of individual developers.

This article provides an overview of the design method as adapted to a particular class of software systems, and suggests guidelines for applying the design method. Related work on software design methodology has been reported in Refs. 1, 2, and 3. The Naval Research Laboratory has reported related work on a real-time system in Refs. 4, 5, and 6. Examples and experiences are presented from our application of these principles to the design of the No. 2 Service Evaluation System (SES), a multiprocessor system performing data acquisition and transaction functions.

We will discuss the key design principles, the proposed design steps and associated documents, the guidelines for preparing each of the documents, and finally, our experiences in applying the principles.

II. A DILEMMA

We are concerned with the dilemma posed by the following two statements:

(*i*) In most software projects coding begins too early. Important design decisions about the functions of the system, the nature of its interfaces, and its maintainability are made as by-products of the coding process and do not receive the conscious attention and review they deserve.

(*ii*) When part of a project's time is invested in a preliminary phase (sometimes called a "concepts phase," "project definition phase," or "specification phase"), one sees little in the way of tangible results. When actual software design begins, the programmers do not use the products of the earlier phase and one has the impression that the time spent was wasted.

These views are held by the same designers at different times in their careers. After an experience without a preliminary design phase, the first viewpoint is espoused with great vigor. After an experience with a preliminary design phase the second viewpoint is held by most participants.

The design method described here attempts to resolve this dilemma by specifying that the preliminary design phases produce a carefully structured set of documents as the main product. The documents are the means to express design decisions, not an afterthought to be

produced after the system development is completed. Since documentation is the main product of the design phases, it is important and must be produced with the same discipline and care with which code is produced.

The principles for organizing the documents are discussed in the following sections.

III. OUR KEY DESIGN PRINCIPLES

The method we are advocating for design and documentation is based on the principles known as separation of concerns[7] and information hiding.[1,2,3] Separation of concerns involves the division of information about a system into clearly distinct and relatively independent parts. A software system design can be better controlled if the information in design documentation is divided in accordance with separation of concerns. The complexity in a software system comes from the number of details that must be considered.[8] To do their jobs, the developers must deal with large amounts of information describing what the system is to do and how their work relates to the work of the other developers. If each design document contains types of information that are clearly distinct and relatively independent from the information contained in other design documents, then the users of the documents can easily determine which document should contain the information of interest.

Ease of change and enhancement of the software system is typically a major objective in adopting a formalized design method. The principle of information hiding can be used to guide the structuring of software to make specific types of changes easy to implement. Information hiding involves encapsulating information likely to change in moderate size software modules. This encapsulation limits the amount of software that must be modified when a change is made. The possibility of future change must be explicitly considered during the design process in order to apply information hiding. One cannot foresee all possible changes; however, by evaluating the possibility of change openly, at least the decisions about what is likely to change are made explicitly and one knows beforehand which functions are likely to be easy to change.

Separation of concerns and information hiding describe the same idea from two perspectives. For example, to fully separate the concerns about the different aspects of a design is equivalent to encapsulating all elements of each aspect, and hence, hiding the information about each aspect. We find viewing some issues from the perspective of separation of concerns to be helpful while other issues are better viewed from the perspective of information hiding. The division of the software documents into clearly delineated areas of coverage is con-

veniently viewed as separation of concerns; whereas, the determination of the information to be contained in a software module is viewed as information hiding.

We discuss a number of applications of these principles in the remainder of the article.

IV. DEFINITION OF TERMS

Before introducing the proposed design method, we define a few terms used in the paper. These terms have been used in a variety of ways in the literature; however, we attach the specific meaning described below.

Software system – A multiperson (and typically multiversion) software development which is delivered and used as a unit.

Input data item – A data item received by the system from a user or an external hardware device or system. An input may be used promptly in the execution of a function, as in the case of a parameter a user enters with the request for a report, or it may be stored in order to influence later operation of the system, as in the case of scheduling information used to control later execution of a function.

Output data item – A data item displayed to the users of the system or sent to an external hardware device or system.

Event – A stimulus to the system causing a function to be performed. Events may be internally triggered upon a change in the state of the system or they may be triggered by a signal from a user or an external hardware device or system. An example of an internally triggered event is the match of the clock time against stored scheduling information that initiates the execution of a function.

Function – The algorithms, rules, or relationships applied by the system in response to events in order to determine the values of one or more output data items and/or the display of the output data items to the user. We do not attempt to fix the size of a function at this point, and discuss both large and small functions. Later, when dealing with module decomposition, we recommend subdividing functions until they are small enough to be developed by one person in a limited period of time.

Module - A piece of software and the associated documentation which together contain all the information about some function(s) or part of a function. Each module is small enough to be developed by one person in a limited period of time—generally one to three months.

Access routine - A piece of software in a module which can be invoked by software in other modules to perform some portion of the module's functions. A subroutine in a data base module which is used by software in other modules to access the data base would be a typical example of an access routine. An access routine is not restricted to being a subroutine. A macro or the top level software controlling a process (which we call the main loop of a process) could also be an access routine.

Process - A set of access routines whose execution sequence is prescribed. The execution of a process may overlap in time with the execution of other processes in the system. In the No. 2 SES we have chosen to restrict the relationship between modules and processes for simplicity. A module does not encompass the main loop of more than one process. This restriction results in some small modules, but it reduces the potential confusion in the relationship between modules and processes.

A module is the basic unit of development and change in this design method. Each module is defined according to the information-hiding principle (i.e., containing all information about some functions) in order to localize the software affected by a change in a function. The limitation to one person doing the development eliminates the need for multiperson communications during the internal development of the module, and the time limitation restricts the amount of work necessary to recode the module in the event of a change.

The usual approach to specifying a function is to describe input, processing, and output, in that order. The above definition of a function leads to function specifications organized around the system outputs. The values and display of systems outputs are specified in terms of algorithms, rules, relationships, inputs, and events. We have found this approach encourages more precise specification of system functions, and reduces the tendency to bias the specification toward a particular implementation.

Table I—Software design documents

Document	Scope
Requirements specification	Everything the software designers need to know about the system.
Module decomposition	The division of the system into modules.
Module dependency	Tabulation of the other modules which each module uses to perform its functions.
Process structure	Groupings of access routines that have prescribed execution sequences.
Resource allocation	System resources used by each module.
Module interface	Everything another programmer needs to know to correctly use the functions provided by the module.
Module design	Description of the internal design of a module.
Test plan	Description of the subset of the requirements which will be tested and the strategies for performing the tests.

V. APPLICATION OF SEPARATION OF CONCERNS TO THE DESIGN PROCESS

We propose dividing the information about the system into the set of documents listed in Table I. This division is based on what we believe are fundamentally separate concerns in the design of a software system. These concerns continue to be relevant throughout the system's life so the documents should be kept up to date.

The relationship of the proposed documents is shown in Fig. 1. The arrows indicate the principal flow of information required for the preparation of each document. Many inputs are, of course, necessary for the preparation of the requirements specification; however, discussion of the many sources of information is beyond the scope of this

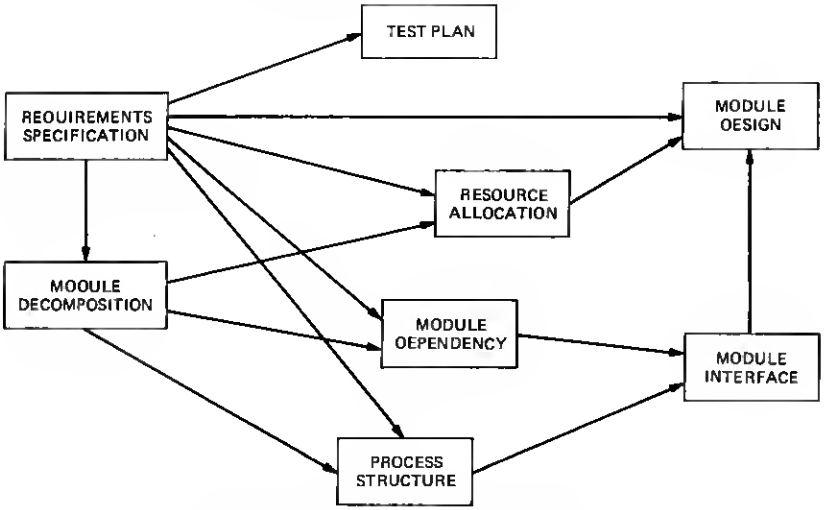

Fig. 1—Relationship of proposed documents.

article. Several feedback paths exist, but have been omitted for simplicity.

The module decomposition, module dependency, process structure, and resource allocation documents collectively constitute an overview of the software structure. Since the first draft of each of these documents can be prepared before any decisions are made about the implementation environment for the system, the overview provided by the documents can provide useful guidance in choosing the processor architecture and operating system.

One could prepare a single-design overview document with chapters dealing with module decomposition, module dependency, process structure, and resource allocation. Similarly, one could have individual module overview documents containing module interface and module design chapters; however, care must be taken to avoid mixing concerns between the chapters in an overview document. We believe the concerns are less likely to be mixed if separate documents are prepared as shown in Table I.

The discussion in the remainder of the paper may appear to represent the design process as a linear progression through the design steps. In fact, experienced developers know feedback to earlier steps occurs repeatedly during the design process. We recognize feedback must occur; however, the feedback should be recorded in the proper document. For example, modifications to the requirements should be found in the requirements specification and not in a note in a module design document. We are aware of the price of adhering to this discipline; however, we feel that the cost of neglecting it is even higher.

We will next present the scope, use, and design considerations for each document. By design considerations, we mean guidelines for the software design associated with the step covered by the document. The guidelines for preparing each of the documents are presented later.

5.1 Requirements specification

5.1.1 Scope

This document, together with the documents it refers to, should contain everything one needs to know to build an acceptable software system. All significant externally visible behavior of the software should be constrained to acceptable alternatives in this document.

5.1.2 Use

The requirements specification can be used both for communicating with the system user and to guide the software design. When the requirements specification has been reviewed by the system users, and the developers and users agree on the contents of the document, it can

serve as part of a contract. Some of the many uses of the document in guiding the software design will be discussed in the following sections.

5.1.3 Design considerations

Many decisions about the functions of a system are made during the preparation of the requirements specification. Recommendations are made later in the paper for structuring the document in a way that encourages systematic resolution of the issues associated with specification of the system functions. The framework of the requirements specification is intended to stimulate addressing requirements issues early in the system design.

Preparation of the requirements specification should start during the earliest stages of a project. The document then evolves as the project proceeds—beginning as a rather sketchy skeleton and gradually filling out until it is complete. Gaps in the requirements specification serve to highlight the open issues.

5.2 Module decomposition document

5.2.1 Scope

The module decomposition document records the division of the software system into modules.

5.2.2 Use

It should tell readers the way the software has been structured and direct them to the appropriate component and its documentation. This document should eliminate any need to search through more detailed documents to find out which one of those documents contains a specific piece of information.

5.2.3 Design considerations

A number of the popular software design methods focus attention on the module decomposition step.[9,10,11] The module decomposition document could be prepared for a decomposition obtained by any of a number of the popular methods; however, since a major objective of the design method is to design for change, we believe module decomposition is best accomplished by applying the principle of information hiding.[1]

Module decomposition according to the principle of information hiding involves systematically hiding in a module all the information about each function defined in the requirements specification. The first step in selecting functions to hide should be to examine the "expected changes" chapter of the requirements specification (see Table II). Any function that is likely to change should be hidden in a module.

After doing the decomposition indicated by the Expected Changes chapter of the requirements, some functions may not yet be associated with modules, and many modules may still be too large. We have approached the further selection of functions to be hidden in modules from two opposing directions. The first involves decomposing the major functions into progressively smaller functions until we judge the implementation effort to be within the constraints we have set for a module. For example, a major function of displaying stored data to the users is broken down into a number of individual reports that can be independently requested. Each report may then be decomposed into a function that controls dialogue with the user, a function to compute output data, and an output-formatting function. As discussed earlier, all major functions can be defined in terms of the information required to determine the output data items associated with the function; e.g., "the prompts to the user," "the output data items required for the report," and "the format of the report."

Continuing subdivision of a complex function may eventually lead to subfunctions that do not directly control an output data item. We introduce the notational convention of intermediate data items to describe the interaction between such subfunctions. Subfunctions and intermediate data items are further discussed in the guidelines later in the article for preparing the data items chapter of the requirements specification. When a function is subdivided, the resulting parts should be chosen so they are likely to change independently.

Another approach to selecting functions to hide in modules is to identify common-use functions from the requirements. One looks for services required repeatedly by a major function or by several major functions. The common-use functions are hidden so they can be changed without affecting other parts of the system. For example, data storage and retrieval services are often required throughout a system. In support of a report generation system, one might identify a common function controlling dialogue with the user, common data base access functions, and a common output-formatting function. Having identified as many common-use functions as possible, one constructs the major functions from combinations of the common-use functions and whatever single-use functions are necessary.

The approach of identifying common-use functions has the advantage of ensuring uniformity in the user view of the system, and it reduces the redundant development of similar functions by several programmers. A disadvantage of this approach is that no single major function can be completed until development is completed on a number of modules hiding common-use functions. On balance, we favor the development of modules that hide common-use functions.

Most experienced software developers will quickly identify a number

of common-use functions that should be hidden in modules. Database functions, device interfaces, output functions, and user interfaces are typical of the functions that will generally be readily identified. During implementation of the modules, the developers may identify the need for additional common-use modules which can be used by two or more developers. Some guidance is available for identifying potential common-use modules;[3] however, good communications among developers continues to be necessary to avoid the development of the same tools by two or more developers.

Access routines in a module may be used by several other modules. For example, portions of a device handler module may be used by a data acquisition module, while other parts may be used by a testing module. Neither of the modules using the device handler would contain any information about the device since that would all be hidden in the device handler.

Module decomposition involves breaking down a multiperson development into individual work assignments; therefore, the initial decomposition must be refined when the implementation effort can be better estimated. If a module is found to require only a small development effort, we generally do not try to merge it with another module since there is not a great deal of overhead associated with having additional modules. If, on the other hand, a module is found to require more development effort than one person can complete within the allowed time limit, then it should be subdivided into two or more smaller modules as described above.

Since the decomposition is based on the functions described in the requirements specification, the decomposition should be independent of the implementation chosen for the modules, with the exception that the amount of implementation effort limits the size of the modules.

The key guideline to keep in mind throughout the decomposition process is to always define a module in terms of the information hidden by the module.

5.3 Module dependency document

5.3.1 Scope

This document specifies for each module which access routines from other modules it must use to perform its function.

5.3.2 Use

The module dependency hierarchy determines which other modules must be available for a module to perform its functions; therefore, the document can be used to identify the modules necessary to provide the required subsets (i.e., the portion of the system to be developed first if time and staffing limitations prevent developing all functions).

The module dependency document is most valuable during the early stages of the design when the development order for modules and the users of each module must be identified in order to prepare and review the module interface documents. Once the module interface documents have been prepared, this document continues to serve as a summary document derived from the module interface documents.

5.3.3 Design considerations

The requirements specification, together with the module decomposition, defines for each module which other modules must be used to perform the required functions. For example, a data acquisition module which is required to obtain data from a device must use the device handler module. Similarly, if the data is to be stored, then the data acquisition module must use a database module. One must systematically examine all of the functions of a module as prescribed in the requirements specification to obtain a list of all of the modules used. No decisions about the implementation of the modules are necessary in order to perform this step. In the case of the data acquisition module, we only need to know that any access to the device must be through the device handler module and any access to the database must be through the database module.

5.4 Process structure document

5.4.1 Scope

This document specifies the groups of access routines having a prescribed execution sequence. The execution of two access routines in the same process are always clearly sequenced, whereas access routines in separate processes can be executed in arbitrary order.

5.4.2 Use

The process structure is a necessary input to the design of the module interfaces since the methods for interfacing between processes are generally different from those used within a process. The groupings of access routines into processes determine which module interfaces are between modules within a process and which cross process boundaries.

The process structure is a major determiner of the potential for exploiting extra processors.

5.4.3 Design considerations

The process structure for a system can largely be defined by determining which functions in the requirements specification can overlap in time and which must be executed in a specified order. A maximum number of processes is obtained if the only modules grouped into

processes are those for which the execution order is prescribed in the requirements specification. All modules for which the execution order is not specified are separated into independent processes. The choice of the maximum number of processes would yield a more flexible design than one with fewer processes; however, the overhead associated with administering processes may cause one to choose a design with fewer than the maximum number of processes.

Additional guidance for defining the process structure is given in Refs. 12 and 13.

5.5 Resource allocation document

5.5.1 Scope

The resource allocation document summarizes the system resources used by each module. The tabulation can include any resource potentially causing a bottleneck in system performance. Resources of concern typically include CPU real time, memory, disk real time, disk space, and communications channels.

5.5.2 Use

This document can be used by module developers to judge the proper level of attention to give to resource usage in the design of each module. If each developer adheres to the resource budget for their module, then the overall system should perform properly.

The document is useful for ongoing tracking of resource usage after the initial design is completed. When enhancements to the system are evaluated, this document can be used to assess potential impact on resource usage.

5.5.3 Design considerations

The requirements specification and module decomposition document provide the basis for determining the frequency of invocation of a module and for estimating the likely resource usage for each invocation. Unfortunately, the initial estimate of resource usage must be based on a rough conception of a possible implementation, and therefore, the estimate may be inaccurate. If a module is likely to consume a large fraction of the system resources, then alternative implementations should be evaluated early in the system design to refine the estimate of likely resource usage.

Substantial effort should be invested in early study of resource allocation. We have seen several projects fail or be severely set back by encountering serious resource usage problems late in the design process. If resource needs are documented early in the project, provision can be made for adequate system resources and for careful design of the modules consuming most of the resources.

5.6 Module interface documents

5.6.1 Scope

Each module interface document describes the aspects of module behavior visible to other programmers using the module. Aspects of behavior visible to the system user are fully documented in the requirements specification and should not be duplicated in the module interface documents. For example, an interface document for a device handler module describes the means for invoking the software, the return values, and any modifications to stored data resulting from invoking the module. No messages exchanged with the device are described.[6]

Everything in the module interface document should be true for all acceptable internal implementations of the module, and should not be biased towards any particular implementation.

5.6.2 Use

The module interface documents settle the agreements between programmers about how cooperating modules will interact. Each module interface document should contain everything another programmer needs to know to develop software that interacts with the module. Clear documentation of agreements between programmers is very important on a multiperson development for smooth integration and ease of maintenance.

5.6.3 Dasign considerations

In order that the module interface documents adequately describe the means of communicating between modules, the implementation environment (e.g. operating system and programming language) must be selected before the documents can be completed.

5.7 Module design documents

5.7.1 Scope

The module design documents are intended to record the decisions made in the internal design of the module. Such topics as data structure design, resource usage, data buffering strategies, subroutine structure, and control logic are appropriate for the module design documents.

5.7.2 Usa

The module design documents are used to guide a review of the software design and to inform future maintainers of the module of the reasons why the particular design was chosen.

5.7.3 Design considerations

Several design methods could be used for the internal design of

modules.[7,9,11] The design method influences programmer efficiency and the maintainability of the module; however, since the design method that we are advocating encourages limiting the size of the modules, an entire module could be discarded and recoded if it proved to be unmaintainable.

The principle of information hiding can be used in the internal design of a module just as it is in the overall system design. If information hiding is applied in the internal design of a module, then the effects of change should be isolated to a portion of the module, and less effort should be required to maintain the module.

5.8 Test plans

5.8.1 Scope

All of the requirements stated in the requirements document are testable, but in practice we can only test a subset due to time limitations. Test plans describe the approach to be used to verify a specified subset of the requirements document.

5.8.1 Use

A separate test group should prepare and execute the test plan. Since the test plans represent only a subset of the total software requirements, the test plan should be maintained as private information within the test group to ensure that software is not written so that it will only pass the test. Even developers with the best intentions may fall into the trap of focusing on the functions to be tested.

5.8.3 Design considerations

The choice of how large a subset is to be tested must be influenced by the potential cost of not finding bugs versus the project limitations in development staff and time. For example, a medical control system could have a very high cost associated with a residual bug in the system. The test plan should explain which potential errors are considered particularly important to detect and what the testing strategy is to detect those potential errors.

VI. DOCUMENTATION PRINCIPLES

Before providing specific guidelines for preparing each of the documents, we will introduce some principles to guide the preparation of any software documentation.

6.1 General principles

(*i*) Write a specification for every document. Five questions should be answered in each document specification:

- Who will use the document?
- What will they use it for?
- What do they know before reading the document?
- What should they know after reading the document?
- What sources are there for prerequisite knowledge?

Note that a document specification is not an outline of the document. Instead, it identifies the audience, the perspective of the audience, and the way that the audience will use the document.[14]

(*ii*) When writing a document, a chapter in a document, a section in the chapter or a paragraph in a section, formulate the questions to be answered before starting to answer them. Writers often confuse organizational issues with issues about the substance of the article; to avoid this confusion, we express the organization in terms of questions rather than answers.

(*iii*) Design documents using the principle of information hiding. Every section of the document should deal with a clearly defined and limited aspect of the system; one should not yield to the temptation to include other "relevant" facts in the same section.

(*iv*) Use formalism to describe design decisions and natural language for introductions, motivation, justifications, etc. Formalisms, when appropriately designed and used, can greatly increase the precision and compactness of a description. Formal descriptions are more easily checked for completeness and consistency. Natural language is preferable to formalism for describing motivational material. There is never a need to describe the same thing both ways.

(*v*) If there are a large number of descriptions containing the same information, restructure the document so common aspects are described only once. It is essential to make the structure explicit or the reader will not know where to find the information that has been pulled out of the individual descriptions to avoid repetition. Repetitious documentation is both time wasting and a cause of errors due to inattentive reading.[15]

6.2 Stylistic rules

(*i*) Eliminate all statements containing little information. If the negation of a sentence would rarely be uttered, the sentence itself communicates very little.

(*ii*) Replace oblique statements with direct statements. Often sentences containing little information are there as indirect ways of saying something else. If something else needs to be said, say it directly.

(*iii*) Avoid saying the same thing twice. If you say the same thing two different ways because neither is perfectly clear, you decrease the clarity because readers will wonder about differences. It is better to spend the time necessary to say it clearly once. However, remember

purpose is different from method, and a decision is different from a reason. Stating the intent behind a design and stating the design is not saying the same thing twice.

(*iv*) When describing a program do not confuse its effects with its intended use. A program may do A and be used to accomplish B, but we often mix A and B in a way that makes it unclear what the program itself actually does.

(*v*) Make the significance of a design decision more explicit by stating the alternatives excluded by the decision. We often read about designs with a "ho hum" feeling because we are not made aware of the significance of the decisions.

(*vi*) Do not justify things in terms of principles nobody could be against. State precisely what pragmatic benefits will result.

6.3 Diagrams in program documentation

Pictures have been hotly debated as a means of documenting programs. If a program is clearly understood, it can be described precisely in terms of predicates and states or in terms of mathematical functions. Pictures tend to be quite imprecise as a means of documentation. On the other hand, pictures are quite useful as a means of introducing someone to a program he does not yet understand. Pictures should be used as introductory material but never as the binding documentation.

When pictures are used, precision in drawing the picture is necessary. Many computer system diagrams are confusing and easily misinterpreted because there is no precise meaning given to the symbols used. Often the same symbol is used to represent a program, a data structure, a hardware device, and a user, all in one diagram. If each picture is accompanied by a legend, there will be less of this confusion.

6.4 Review procedures

Effective review of the documents serve to verify the correctness of the documents and to ensure that they are understandable. The following guidelines can help one achieve effective document reviews.

(*i*) The selection of the reviewers for a document can be approached at the following levels depending upon one's objectives.

(a) The user of the software is an obvious choice for a reviewer. This would be the system user for the requirements specification, and the software developer who will use the module in the case of a module interface document. The user has a clear interest in the proper operation of the software, and hence, has a reason to do a thorough review of the document.

(b) A developer other than the one who prepared the document can be given the work assignment of reviewing the document. This so-called "buddy system" can result in someone else in the

development group who is responsible for the correctness of the document and who is prepared to defend it. An additional benefit of the "buddy system" is the cross-knowledge gained within the development group. This cross-knowledge can be helpful when task reassignments are necessary.

(c) A person outside the project can be brought in to review the document. This reviewer will uncover omissions presumed to be common knowledge by those closer to the development. The outside reviewer is also a good choice for reviewing the overall set of documents for consistency.

(*ii*) The reviewer should be asked to examine the document from a specific perspective. For example, an expert in the system outputs should be asked to verify that section of the requirements specification. A questionnaire can be used to ensure reviewer will consider specific issues. Such a questionnaire should be prepared by the person who is directly concerned with the correctness of the document.

(*iii*) The reviewer should be asked to provide input in a comments section of the document. The reviewer should sign off on the document and note the areas of the document with which they were chiefly concerned. A record is then available of who has examined the document. The reviewer can be consulted later if issues arise that they may have considered.

6.5 Inclusion of justification material in the documents

Arguments can be made both for and against the inclusion of justification material in the documents to record why decisions were made. On the one hand, inclusion of a justification section in each document encourages the writer to record the reasons for making each decision at the time the decision is made. The reader is also more likely to read the justification material if it is included in the primary documents.

On the other hand, justification material can be quite verbose and its inclusion can swell the size of the document to the point that it becomes unwieldy, and the potential users of the document are discouraged from reading the document by its sheer bulk. The use of separate justification documents (referred to in the primary design documents) encourages clear separation of the concerns between what was decided versus why it was decided.

Faced with this dilemma, we have chosen to use separate justification documents for all of the documents, except the module interface and module design documents. The documents dealing with the whole system are quite large—particularly the requirements specification. Inclusion of justification material in these would make them excessively bulky and forbidding. The individual module interface and

module design documents are typically only a few pages in length so the inclusion of justification material does not make them excessively large. A principal part of the module design document is, in fact, an explanation of the strategies used in the design.

VII. DOCUMENT PREPARATION GUIDELINES AND EXAMPLES

In this section, we provide preparation guidelines for each of the documents.

The guidelines and examples for the requirements specification are more detailed than for some of the other documents; however, the other documents may be of equal or greater importance for a particular project, and some of the other documents may require more effort to prepare. For example, module interface and module design documents are prepared by each software developer, so the collective effort in this area is quite large.

7.1 Requirements specification

The guidelines we have evolved for preparing the requirements specification for the No. 2 SES are based on a model project to prepare requirements for a real-time system.[4,5] The transaction-oriented nature of the No. 2 SES has motivated us to shape the guidelines to be more appropriate for our type of system.

We believe the requirements specification is most effective if it is a concise reference document. Formalisms are used wherever possible and tabular organization is frequently used. These techniques aid us in making the document concise. A concise document may require some additional effort for first-time readers to familiarize themselves with the formalisms and background material; however, the concise format is more efficient for day-to-day use, it eases updating of the document, and it encourages precision in the specification of requirements.

We organized the document into ten chapters which separate the concerns about the external behavior of the system. The chapter organization we have used is shown in Table II.

7.1.1 Introduction

The introduction should provide a guide to reading the document rather than an introduction to the system. Reference can be made to a separate system description for an overview of the system. We include a discussion of the organization of the document. Formalisms are explained and examples of the formalisms are given.

7.1.2 Input and output data items

The input and output data items are specified in several tables and

Table II—Chapter organization

Chapter	Contents
1. Introduction	A guide to using the document.
2. Input and Output Data Items	Definition of the input and output data items presented to the user and/or to external devices or systems.
3. Communication Protocols	Details of communications with hardware devices, software systems, and users. The user command syntax may be included here since it is a protocol.
4. User Transactions and Reports	Specification of the user interaction with the system, plus all scheduled and spontaneous reports generated by the system.
5. Performance Requirements	Constraints on how functions must be performed. We include timing, concurrency, accuracy, and storage considerations in this chapter.
6. Response to Undesired Events	What the software must do when undesired events occur.
7. Fundamental Assumptions	Characteristics of the system that are not expected to change.
8. Expected Changes	Changes expected or planned for future releases.
9. Required Subsets	Description of one or more subsets of the functions which would still constitute a useful system.
10. Glossary of Acronyms and Terms	Explanation of the acronyms and technical terms associated with the system.

supporting sections within this chapter. This chapter corresponds to a data dictionary.

Examples in Tables III, IV, and V illustrate the techniques we have used to specify the data items. These examples are arranged around a user transaction in the No. 2 SES needed to display some of the data stored about entities (telecommunications switches). The display consists of a set of output report data items.

We introduced the concept of data types to aid in the specification of the data items. Two criteria are applied to determine whether two data items are of the same type.

(*i*) The data items have the same set of values.

(*ii*) It is meaningful to use them in an assignment statement. For example, even though a computer identifier and a data link identifier might have the same set of values, using them together in an assignment statement would not be meaningful.

We bracket an item by "+" to indicate it is a data type. Our text processing system is used to audit the data types to ensure that every data item has a valid type and every type is used in at least one data item. Sample data types are presented in Table III. An enumerated type is a set of values. A list is a one-dimensional array.

Input data items are grouped into two classes—user inputs and device inputs. Output data items are grouped into three classes—report data items, interactive messages (errors, help, prompts, and positive feedback), and outputs to devices.

Input and output data items are bracketed by "/" and "//", respectively. All references to the data items use the bracketed notation.

Table III—Data types

Type	Values	Description
+boolean+	enumerated	boolean (two values)
	YES	yes
	NO	no
+ent-no+	integer, range (1–999)	entity number
+ent-state+	enumerated	entity state
	$NOT-DB$	not in database
	$READY$	ready
	$OFF-MAN$	off manual (user action)
	$OFF-AUTO$	off automatic (by program)
+list-4+	list of integers size 4 entries	list used for many reports
+nsc+	integer, 4 digits	network service center number

Wherever a bracketed item appears in the document, the reader can readily recognize that it is an input or output data item. When we change a data item, our text processing system is used to search for all occurrences of the bracketed item.

Separate tables are prepared for input and output data items. A description of each data item is provided in these tables, and the data type is specified. The specifications of the functions controlling each output data item are identified in the table for output data items. We have categorized the No. 2 SES functions as either user transactions or data acquisition functions, and have grouped the specifications of the functions into two separate lists. The %A-B% notational convention shown in Table IV is used to point into the two lists of function specifications. The A number points to the specification of the user transaction controlling the output data item, and the B number points to the specification of the data acquisition function. If both A and B are nonzero, then the output data item values can be set by either a user transaction or a data acquisition function; e.g., the entity state data item, //ENT-STATE//, in Table IV can either be set by the user or by a data acquisition function responding to an error event.

Some functions are made up of several parts that may change separately. Such functions should be described in terms of two or more subfunctions each of which is likely to change as a unit. To describe the communications between individual subfunctions, we have intro-

Table IV—Output data items

Data Item	Description	Functions	Data Type
//ENT-CLLI//	entity's text identifier	%1-0%	+char(13)+
//ENT-COM-PT//	common evaluation done on entity?	%1-0%	+boolean+
//ENT-LOOPMAX//	maximum loop on entity	%2-0%	+loop-no+
//ENT-NO//	entity's number	%1-0%	+ent-no+
//ENT-NPA//	entity's NPA	%1-0%	+npa+
//ENT-STATE//	entity's state	%3-8%	+ent-state+
//CALL-DISP//	call disposition	%0-12%	+disp+

duced the notational convention of intermediate data items (bracketed by !). An intermediate data item is not visible to the user and serves only as a notation for the output of one subfunction that is, in turn, used as an input to another subfunction.

The No. 2 SES function of determining the disposition of a customer call attempt is subdivided into two subfunctions. The first determines the initial disposition !INIT-DISP! by analysis of voice signals. The second subfunction uses stored information about the data source and the value of !INIT-DISP! to determine the final call disposition //CALL-DISP//. The two subfunctions are likely to change independently so they are hidden in different modules.

Intermediate data items are specified in a table similar to that used for output data items.

7.1.3 Communications protocols

The communications required with external hardware devices and software systems are specified in this section. External hardware devices include devices used for data acquisition, control, and/or display. If the communications with an existing device or software system are fully documented elsewhere, then the appropriate document can be cited.

The user command syntax rules can also be specified here because the syntax rules can be viewed as a protocol; however, the detailed command semantics should be specified in the chapter on user transactions.

7.1.4 User transactions and reports

All functions of the system visible to the user are specified in this chapter. These functions include computer operations, data base interactions, maintenance, user requested reports, scheduled reports, and spontaneously generated reports, such as equipment failure alerts. These functions are defined in terms of the input and output data items defined in the data items chapter. The same data items may appear in many reports.

A sample specification of a user transaction to obtain an output report is given in Table V. Some words of explanation may be needed to interpret the notation. The User Data Entry specifies the user input required to produce the report. The user enters the command "display entity-list" and selects which entities are desired. The default value for the entity selection is "all" entities. The output report is the collection of data items listed under Output Values. These are output for each entity displayed. The Transaction Effects section describes any changes to the state of the system or modifications to stored data resulting from performing the transaction; therefore, in this example,

Table V—Output report specification

```
Transaction name: entity list
User data entry: display entity-list /ENT-SELECTION/ = all
Output values:
  //CH-NO//
  //ENT-CLLI//
  //ENT-COM-PT//
  //ENT-NO//
  //ENT-NPA//
  //ENT-NSC//
  //ENT-STATE//
  //SCA-Port//
Transaction effects: none
Error messages:
  a. type error +ent-select+
  //E-ENT-SEL//
  b. constraint error +ent-select+ (entity not in database)
  //E-NO-ENT-SEL//
```

the Transaction Effects are "none" because this function simply reads the database and leaves no trace. The Error Messages section lists all messages specific to this transaction. The type error can be detected by examining the input data item itself, whereas constraint errors must be determined by checking the input data items against data stored in the system. The purpose and use of this report are not discussed here—that information is contained in the user guide.

7.1.5 Performance requirements

The preceding chapters of the requirements used the narrow definition of a function as being what the system was to do. The considerations of timing, concurrency, data volumes, data retention, and data accuracy are reserved for this chapter. A separate chapter is provided for these considerations because the requirements on what the system is to do may change separately from the performance requirements.

The process structure selected for the system must permit satisfying the sequencing and concurrency requirements described in this chapter.

7.1.6 Response to undesired events

Undesired events (UEs) prevent the software from performing the desired functions. Undesired events may be caused by input data errors, computer hardware malfunctions, or software errors. The number and variety of things to go wrong are quite large, and one has difficulty anticipating all possible problems when writing the requirements. One can begin by documenting all known UEs together with the desired response to each of the UEs. As the system is developed, additional UEs will be identified, and can be documented in this chapter.

The key objectives are (*i*) consciously consider the desired response

to each UE, and (*ii*) document all UEs and responses to the UEs in one place.

7.1.7 Fundamental assumptions

This chapter consists of the list of functions and subfunctions that are not expected to change during the life of the system.

Discussions between the users and developers about which functions are not likely to change can be a useful part of the review of the requirements. These discussions will often involve challenges to the fundamental assumptions and may result in moving several of these functions to the next chapter. If the users wish to change functions appearing in this section after the system has been developed, they can expect such a change will require a large development effort since the developers did not have a reason to make it easy to change. Note that one does not try to make a function difficult to change. It will naturally become difficult to change if all information about the function is not carefully hidden in a module.

7.1.8 Expected changes

This chapter complements the fundamental assumptions chapter by providing a list of functions that are expected to change. The lists of functions in this chapter and in the chapter on fundamental assumptions should constitute a complete list of functions since a function is either likely to change or not.

Since many functions are likely to change at some point in the lifetime of a system, ranking the expected volatility of these functions may be helpful. Changes to some functions may already be planned for a future release of the software. Changes to a second group of functions may not be planned, but from historical data one knows functions of this type have often been subject to change. One may have no reason to expect changes in a third group of functions, but, on the other hand, there may be no firm reason to expect them to be stable. The additional cost of designing, with the expectation that most functions may change at some point, is modest in relation to the cost of later changing a function for which no thought had been given to possible change.

Functions that are likely to change should be carefully considered when the decomposition into modules is performed. At that stage, one should ensure that all information about each of these functions is completely contained in a module.

7.1.9 Required subsets

The functions of the system should be analyzed to determine what would constitute a minimal useful system. This minimal subset should

be the first part of the system to be developed. If delays are encountered in developing the software, then a useful subset of the system can still be delivered on a timely schedule.

This chapter should define the minimal subset, plus any larger subsets which would provide additional valuable functions.

Developers are often under pressure to start development before the requirements are fully defined. If moderate risk can be taken, development can, in fact, begin once certain critical parts of the requirements are completed. If the functions in a minimal subset are defined and the performance requirements, fundamental assumptions, and expected changes associated with the minimal subset are defined and reviewed, then one can start the development of the minimal subset without excessive risk of wasting effort. A continuing source of risk arises from the possibility that the performance requirements for the full system will be more demanding than for the minimal subset. Common-use modules should be developed to accommodate the performance demands anticipated for the full system.

The people on a small project will often have excellent informal knowledge of the requirements before the formal document is written. In such a situation, the other design steps can be started, while the requirements specification is being written; however, issues that were thought to be resolved are often revealed to be incompletely defined when the attempt is made to write them down.

7.1.10 Glossary of acronyms and terms

This chapter is, of course, useful in supporting all of the document; however, it is particularly helpful in expanding the descriptions of the data items.

7.2 Module decomposition document

In the module decomposition document, we list the modules produced in the decomposition phase, and state what information is hidden in each module. Since a concise document is desired, we do not include any discussion of the strategy used to obtain the decomposition; instead, we refer to a separate justification document.

The modules are grouped into major classes to assist in locating a module dealing with a particular type of information and to assist in reviewing the decomposition for completeness. Most systems will have at least three classes of modules that hide (*i*) hardware information, (*ii*) user visible behavior, and (*iii*) software design decisions. We have found a somewhat larger number of module classes to be useful for the No. 2 SES. Our module classes are as follows: Database, Device Interface, Data Acquisition, User Input/Output, and Maintenance.

To review the completeness of the decomposition, we check to

ensure that the information about each function in the requirements specification is stated to be hidden in some module.

A portion of our module decomposition document is shown in Table VI. Since the modules represent individual work assignments, we have found the document to be quite useful in tracking the progress of the work; therefore, we identify the author and reviewers of each module (the author and reviewers are not shown in Table VI to save space), and we have indicated the current status of the module. The abbreviations in the status fields are as follows:

NW	Module interface document has not been written.
MS	Module interface document has been written.
MSR	Module interface document has been reviewed.
DD	Module design document has been written.
DDR	Module design document has been reviewed.
C	Coding of the module has begun.
CR	Code for the module has been reviewed.

7.3 Module dependency document

The module dependency document should list all of the modules in

Table VI—Module decomposition document

Class	Module	Status	Information Hidden
Database	—	—	Modules providing access to the stored information.
	Call-record	CR	Storage and retrieval of call data.
	Bureau	CR	Storage and retrieval of bureau data.
Device interface	—	—	Modules providing communication to the call acquisition hardware.
	SCA*-handler	CR	Communications with SC/A devices.
	VDAS†-handler	NW	Communications with VDAS devices.
Input-output	—	—	Modules providing the user-required inputs and outputs.
	Term-interface	CR	Syntax rules for the user terminal command and feedback.
	Bureau activity	CR	Report summarizing system activity.
	DB-builder	C	The means for the user to alter contents of the No. 2 SES databases.
Data acquisition	—	—	Modules associated with the acquisition of call records.
	CR-generator	C	Control of the acquisition of call records.
	Classify-call	C	The computation of call dispositions from input voice and call data.
	CCT-sched	C	Scheduling the acquisition of calls from entities.
	Cr-proc	C	Control of the processing of call records.

* Signal converter allotter.
† Voice data systems.

the system, and for each of the modules a secondary list should contain all of the modules used by each module. An example module dependency document is shown in Table VII.

7.4 Process structure document

The process structure document should list all of the processes in the system and indicate which module encompasses the main loop of the process. Recall, we have restricted the scope of modules so no module encompasses the main loop of more than one process. We give the process the name of the module which encompasses the main loop.

The modules containing the main loop of a process are identified in Table VII. In fact, this simple table could serve as a process structure document; however, as we discuss later in the experiences section, the process structure document for the No. 2 SES includes a summary of interprocess communications.

7.5 Resource allocation document

The resource allocation document should contain a list of all modules and the amount of resources allocated to each. The total resources consumed when the module is invoked should be recorded including the resources consumed by any subordinate modules used.

When the module design documents are available, the resource allocation document can be derived from information in the module design documents, and the resources used by each access routine can be included. Thus, just as the module dependency document becomes a summary document once the module interfaces are written, this document also becomes a summary document once the module design documents have been prepared.

Table VII—Module dependency document

Module	Other Modules Used	Process Main Loop
Call-record	—	—
Bureau	—	—
SCA-handler	—	—
VDAS-handler	—	—
Term-interface	Bureau	—
Bureau-activity	Call-record	X
	Bureau	—
	Term-interface	—
DB-builder	Bureau	X
	Term-interface	
CR-generator	SCA-handler	X
	CCT-sched	—
	Classify-call	—
Classify-call	—	—
CCT-sched	—	—
CR-proc	Bureau	X
	Call-record	—

Table VIII—Standard form for module interface documents

Section	Contents
Module name	Name of module.
Author	Name of author.
Reviewers	Names of reviewers.
Information hidden	List of functions for which all information is contained in the module.
Access routines	The process and/or subroutines invoked by other modules to perform the functions provided by this module. The input parameters and return values for each access routine are defined using the conventions of the chosen programming language.
Effects on stored data	The stored data items modified by the invocation of the access routine are tabulated for each access routine. All changes in the system resulting from the invocation of each access routine should be recorded; therefore, the effects of all other modules used by the access routine to perform its functions must be noted.
Undesired events	Undesired events UEs occur when an access routine is not able to perform the requested function. All potential UEs for each access routine are listed and the return value is specified for each UE.
Other modules used	List of access routines in other modules that must be invoked in order for this module to perform its function.
Design issues	Discussion of the design issues which were considered in choosing the access routines to be provided, in choosing the input parameters and return values, and in defining the UE responses. Typically, the discussion of the choice of the UE responses is a major part of this section.
Review comments	The reviewer's comments and sign-off. This is the only section of the document that the reviewers may edit.

7.6 Module interface documents

Each module interface document should contain all the information another programmer needs to know to use the module. The document should specify how to invoke the module, what functions are performed by the module, and what return values and error indications are provided.

Since one of these documents will be prepared for each module, we used a standard form for the document to ensure that the same information is available for each module and to make it easier to find information in the document. A description of the contents of each section of our standard document is shown in Table VIII.

7.7 Module design documents

Each programmer should prepare a module design document before the code is written. The document deals only with the implementation of the functions of the module. The strategies the programmer used in the design are discussed. Typical topics include data buffering strategies, resources usage, subroutine structure, UE handling strategies, and program control flow. Pseudocode is used to show the control flow. Pseudocode is more readable than the "prose programs" often written when a programmer attempts to document what their software does.[16]

Design documents for modules that invoke a subordinate module should identify it, but not describe the internal design of the subordinate module.

7.8 Test plan

The test plan should list the tests to be performed together with the planned order of testing, testing strategy, and test environment. The test should be prepared directly from the requirements specification rather than from any lower level design documents. The test descriptions should refer to the requirements for the functions to be tested rather than paraphrasing the requirements since such paraphrasing may introduce subtle differences between the test objectives and the requirements.

VIII. DESCRIPTION OF THE NO. 2 SES DEVELOPMENT ENVIRONMENT

To provide the reader with some perspective on our experience with the design method, we will discuss the purpose, resources, and development environment of No. 2 SES.

The No. 2 SES collects data on the quality of service offered to the users of the telephone network. More specifically, it collects data on whether a customer-dialed call attempt succeeds or fails, and if it fails, the failure type. This data on network performance is the basis for an overall assessment of the adequacy of equipment provisioning and maintenance.

The No. 2 SES architecture, illustrated in Fig. 2, consists of a central

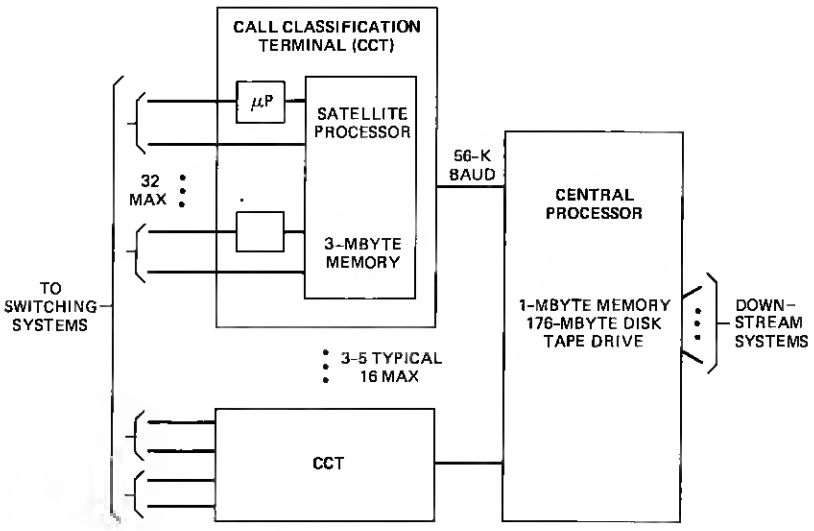

Fig. 2—No. 2 SES architecture.

processor and a number of satellite processors. The satellite processors are used for a signal recognition task requiring extensive computation. The satellite processors are, in turn, each supported by 32 microprocessor systems that extract data from analog telephone signals.

The development environment and schedules for the No. 2 SES project have much in common with a number of operations systems developed at Bell Laboratories over the past eight years. The system uses an enhanced version of the *UNIX** operating system and is programmed in the C language. The development group is modest in size—on the order of ten people. The development schedule is typical of the first development cycle of many operations systems. Feasibility was examined with a small staff beginning in 1978. The definition of functions and architecture were done in 1979, and the development group was fully staffed. Most of the coding was done in 1980, and the first field system became operational early in 1981.

No relaxation in the schedule nor increase in staffing was provided to aid the prove-in of the new software method. The effort invested in generating additional documentation was offset by effort saved during the system integration largely due to the clear expectations between developers fostered by the use of module interface documents. One of the authors was employed as a consultant between November 1979 and July 1980, and another worked full time on the requirements specification.

IX. EXPERIENCES IN APPLYING THE DESIGN METHOD TO THE NO. 2 SES PROJECT

9.1 General comments

Since we are writing this paper shortly after the No. 2 SES became operational in the first field application, we cannot present a full retrospective evaluation of the process; however, we will review some of our experiences thus far in applying the design method.

The development environment for the No. 2 SES has, of course, shaped our experience in using the design method. The environment of a small staff working against a tight development schedule offers advantages of flexibility and easy communication throughout the group, but on the other hand, there is little time or staff available to prepare detailed plans or to provide detached review and testing support. Most of the projects with a small staff with which we have been associated in the past have taken advantage of the easy communications within the group, and have correspondingly minimized the amount of documentation prepared. Such projects have often met

* Registered trademark of Bell Laboratories.

their initial objectives, but have been costly to maintain over their lifetime.

We were handicapped by adopting the principles after the development was well underway. We had to learn how to apply the principles, while the development was proceeding under the constraint of a fixed-project schedule. Much of the additional effort we have incurred in using the design method has been the result of the inefficiency of trying to learn the method, while the development was in progress. Hopefully, this article will help the reader understand beforehand what is involved in adopting this design method so the learning phase can precede the development rather than being concurrent with it.

9.2 Requirements specification

9.2.1 Relationship between the user guide and the requirements specification

We prepared a draft user guide late in 1979, and used it for a review of the proposed system features with an advisory panel of prospective Bell System operating company users. With the draft user guide as a starting point, we prepared a requirements specification in the first half of 1980. The first step in preparing the requirements specification involved recasting the general feature descriptions contained in the user guide into the more precise format described in this article. Many decisions were required to make the general descriptions more precise. Additional material was then prepared for the chapters on performance requirements, undesired events, fundamental assumptions, expected changes, and required subsets.

The overlap of the user guide and requirements specification has been a continuing source of concern. We now see how to have common-source text files form the core of both the requirements specification and the user guide. The requirements specification is divided into those parts visible to the user (e.g., reports) and those parts not visible to the user (e.g., communication protocols). The user guide is constructed by augmenting the user visible portion of the requirements specification with descriptive material to explain the intended uses of the functions. The use of common source files for both documents avoids the duplication of information that makes documents so difficult to keep up to date. We are only now starting to implement the use of common source files for the requirements specification and user guide.

9.2.2 Preparation of the document

Our late start on preparing the requirements specification dictated that it be written in parallel with the other design steps. During the preparation of the document, we depended upon the general knowledge

of the requirements within the group and upon the draft user guide which described most of the output reports.

The sections on output data items and reports were prepared first. The database was defined from these sections. The input data items and user transactions were defined next. The communications protocols with external devices and systems were documented elsewhere, so preparation of these sections did not have high priority. The user command syntax was defined after the development was well underway.

Considerable time was consumed in choosing the organization for the document and the formalisms to be used. We now believe the basic chapter organization proposed here is sufficiently general to satisfy the need of a broad range of developments with minimal modifications. Much time was devoted to selecting the formalisms for describing the data items. Time can be saved by starting with simple formalisms to describe data items, such as the table descriptions illustrated here. If the simple formalisms prove to be cumbersome and verbose for a portion of the data items, then additional formalisms can be introduced to handle just the troublesome items. Our selective use of intermediate data items is an example of this approach. Similar selective use of formalisms is appropriate for user transaction and report descriptions. The more sophisticated notational conventions, such as modes and event tables in Ref. 4, yielded more concise descriptions of real-time functions than the simple formalisms we have used.

The size of the requirements specification is a major concern of many people who are considering this design method. Concern about size is appropriate when deciding how to staff the task of preparing the document, and when considering subdivision of the document; however, size should not be a consideration when deciding whether to prepare a requirements document. We do not know of a good alternative to adequately document requirements. There are many examples of projects that experienced serious trouble because they did not have well-defined requirements.

Issue 2 of the requirements specification for the No. 2 SES contains about 250 pages. The specification of 115 user transactions and reports occupies about 150 pages. About 50 pages are required to describe 800 input and output data items and 80 data types. The remaining 50 pages is mostly text. The number of user transactions and data items required for a system is a useful indicator of the potential size of the requirements specification.

9.3 Module decomposition

We established the module decomposition with surprising ease and unanimity among the people involved in the task. This decomposition

has remained substantially intact through the rest of the development. Most of the later changes have involved the definition of additional modules as the requirements have been refined in areas that initially were vague.

Provided the requirements are clearly understood and the decomposition is approached by asking questions about what functions of the system should be hidden in modules, then we believe most people will generate similar module decompositions. The chief difference we have seen in the results of several people doing a decomposition is the degree to which the system should be broken down; i.e., should some function of the system be in a single module or should the function be divided into two or more modules. For example, we had no difficulty agreeing database access routines belong in a different module from data acquisition tasks; however, we could not definitely determine whether all database functions should be in one module or whether we should have several database modules. The appropriate size for a module is difficult to estimate early in the design; fortunately, it is easy to later decompose a module into two or more smaller modules if closer examination of the implementation indicates too much work is involved.

9.4 Module dependency

The module dependency for the No. 2 SES was rather simple. Only the database and user interaction modules were extensively used throughout the system. Most of the other modules had one or two users. If we were developing an operating system rather than an application based on an existing operating system, we might have found a much more complex module dependency. Since most commonly used utilities for our system are provided by *UNIX*, we only needed to develop a few common-use modules.

9.5 Process structure

We chose a process structure allowing very near the maximum concurrency permitted by the requirements. Most of the small processes resulting from the maximum partitioning reside on the central computer. The central computer has ample resources available in the initial versions of the system so the overhead of administering the additional small processes is acceptable, and the ease of maintaining the small processes is valuable.

The multiprocessor architecture of the No. 2 SES has caused us to have more complex interprocess communications than would have been necessary for a single processor system. Because of the complexity of the interprocess communications, we have found the inclusion of an overview of all interprocess communication in the process structure

document to be useful as an aid in introducing people to the system design. This overview is derived from the module interface documents.

In some cases, implementation considerations have caused us to use two *UNIX* processes to perform the functions of one logical process. Interprocess communications sequence the execution of the two processes as prescribed in the requirements. For most purposes, these two *UNIX* processes can be considered to be one logical process.

9.6 Resource allocation

A small number of modules in the No. 2 SES consume most of the system resources, and we were careful to track the resource usage of these modules. We did not recognize the need for a resource allocation document until well into the development so the tracking has been informal. If resource usage had been more uniformly distributed among the modules, we would probably have been motivated to prepare a resource allocation document earlier in the development.

9.7 Module interface

We have prepared a module interface document for each of the modules in the system using the format illustrated earlier. These documents have been quite valuable in coordinating work among developers on the project. Our experience confirms the expectation that the use of module interface documents reduces the effort required for system integration. Misunderstandings about interfaces are exposed during system integration. Since we had documented and reviewed the interfaces before coding started, we discovered fewer misunderstandings during system integration.

9.8 Module design

These documents have been useful for guiding the review of the design. We have not used a standard format for these documents, partially because we did not have a clear idea of what a good format would be. The format for the documents written by the developers has tended to converge during the course of the development so we could probably specify a suitable standard format now.

9.9 Test plan

Developers test their own module, and a small integration and test team tests the overall system. A testing strategy was established early in the development, and more recently, we have devised a detailed test plan. The minimal subset of the system was developed first, and we have used the subset to provide the test environment for the remaining features in the system.

X. APPLICABILITY OF THE DESIGN APPROACH TO LARGER AND SMALLER PRODUCTS

We believe the design method described here can be used effectively on both small and large projects. Resistance can be encountered from people on small projects who are often able to learn most of the requirements and design decisions, and therefore, do not see the need to generate documentation containing the level of detail we have described here. To accept the need for careful documentation one must recognize that most software systems must be maintained for a number of years, and the original developers generally move on to other projects. If the original developers do not adequately document the design, replacement people find the maintenance of the system increasingly difficult as the reasons behind undocumented design decisions are lost.

The need for careful documentation is more readily accepted by people on a large project; however, we have observed cases where people on large projects have overreacted by specifying the generation of redundant documentation that has been a burden to the project.

People on a large project are likely to recognize that a precise specification of requirements is essential to guide development and testing. Module interface documents are particularly important for a large project since the agreements between developers become much more complex as the number of developers is increased.

A large project is often subdivided into several subsystems in order to aid project management. All of the design steps described in the article could be applied to each subsystem. The requirements specification for a subsystem would include functions that are external (visible to the system user) and others that are internal (visible only to the developers of other subsystems). To obtain a complete view of the user visible functions, the text files describing the external functions of each of the subsystems could be combined into a single document. Information hiding should guide the decomposition of a large system into subsystems.

XI. USING THE DESIGN METHOD AS THE BASIS FOR PROJECT MANAGEMENT

The framework provided by the design documents can be used as the basis for project management. The agreements with the user about the functions of the system are embodied in the requirements specification. The basic development unit is the module—a work assignment for one person for a limited period of time. The agreements between developers are recorded in the module interface documents. The order in which the modules are developed is determined from the combina-

tion of the required subsets chapter of the requirements specification and the module dependency documents.

Several additional planning and tracking tools (e.g., PERT charts) are needed to aid project management; however, the additional tools should use the work units and agreements specified in the design documents as building blocks. For example, a PERT chart displaying development activities should use modules as the basic development units and the completion of required subsets should be major milestones in the development.

We have used the design method as the basis for managing the development of the No. 2 SES. The module interface documents have been particularly valuable. With the module interface document agreed upon before internal design of the module begins, the developer is much freer to work independently on the development of the module. The developer only needs to negotiate with other members of the development group if a change is required in the module interface. If the supervisor and developer agree on a work plan for developing the module, then the developer is free to execute the work plan without continual involvement of the supervisor or other group members. This autonomy fosters a high level of professionalism and a sense of personal responsibility.

XII. MAINTENANCE OF THE DOCUMENTS

We have used a concise format for most of the design documents. Justification material has been separated into supporting descriptive documents with the exception of the module interface and module design documents. Lists of modules or data items make up much of the other documents.

The concise format of the documents should ease updating and checking for consistency. Automated text processing and static code analysis tools are readily available to reduce the amount of the manual document updating. We have used *UNIX* text processing capabilities to check for consistency and usage of data items.

Several tools are available to extract dependencies from source code. Use of these tools could ensure that the design documents were consistent with the source code. We have not yet adapted these tools for use with our documents; however, we hope to use them in the future. The documents that could be automatically checked for consistency with the code include the module decomposition, module dependency, process structure, and module interface documents. The communications protocols, data items, and user transaction chapters of the requirements specification could also be similarly checked.

The resource allocation document must be updated from system

resource usage measurements. Justification documents including the module design documents must be manually updated and reissued periodically.

XIII. CONCLUSION

The principle of separation of concerns requires the division of the design information into clearly distinct and relatively independent documents. These design documents are the main products of the initial design process and, therefore, are the instruments for recording and communicating design decisions. The documents are to be kept up to date throughout the lifetime of the project so one should be able to find current information on any aspect of the software design by examining the relevant document.

The principle of information hiding is used to guide the internal design of the software. The functions of the system that are expected to change are hidden in modules in order to minimize the amount of software affected by a change in these functions. Explicitly designing for change is very desirable for systems like ours that are expected to evolve over a period of years. By giving explicit consideration to the possibility of change, we have identified many potential areas of change. Even so, changes are sure to be proposed that we did not anticipate. We will at least know immediately whether a proposed change is likely to be easy to implement or not.

No design method will prevent one from making bad design decisions; however, the framework provided by the design documents encourages systematically answering a comprehensive set of questions about the system. This process of answering questions may uncover issues often overlooked until late in the development process when they have a costly impact. The design method does not alter what issues must be resolved, but it does change when and how the issues are decided and documented. For example, often requirements issues are not decided explicitly; instead, in the course of the coding, the programmer comes to a point where a decision about external behavior must be made for their work to proceed. They either consult someone or make a private decision. With the design method described here, when a requirements issue is recognized, it is stated in the requirements specification, and the choice of the desired external behavior is made openly. When the choice is made openly, the alternatives will often be more carefully considered. More effort may be invested in making the decision; however, time spent making a careful decision is generally well spent.

The effort required to apply these principles to the development of the No. 2 SES has been accommodated within the development interval originally allocated. The immediate benefits we have gained are (*i*)

control of the design process and (*ii*) smooth system integration. In the future, we hope to be able to implement expected changes at low cost.

XIV. ACKNOWLEDGMENTS

Many members of the No. 2 SES Development Group have helped shape these procedures to be more effective for our application. Contributions worthy of special note were made by David T. Johnson in defining the module interface document and by Maureen Ahern in influencing the module design documents.

REFERENCES

1. D. L. Parnas, "On tbe Criteria To Be Used In Decomposing Systems into Modules," Commun. ACM, *15*, No. 12 (December 1972), pp. 1053–8.
2. D. L. Parnas, "Use of Abstract Interfaces in the Development of Software for Embedded Computer Systems," Naval Research Laboratory, Washington, D. C. 20375, NRL Report 8047, 1977.
3. D. L. Parnas, "Designing Software for Ease of Extension and Contraction," Proc. of the Third Int. Conf. Software Engineering (May 1978), pp. 264–77.
4. K. Heninger et al., "Software Requirements for the A-7E Aircraft," Naval Research Laboratory, Washington, D. C. 20375, NRL Memorandum Report 3876, November 27, 1978.
5. K. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and Their Application," IEEE Trans. Software Engineering, *SE-6* (1980), pp. 2–13.
6. K. H. Britton, R. A. Parker, and D. L. Parnas, "A Procedure for Designing Abstract Interfaces for Device Interface Modules," Proc. Fifth Int. Conf. Software Engineering, ACM Order No. 592810 (1981), pp. 195–206.
7. E. W. Dijkstra, *A Discipline of Programming*, Englewood Cliffs, NJ: Prentice-Hall, 1976.
8. H. D. Mills, "How to Make Exceptional Performance Dependable and Manageable in Software Engineering," Proc. COMPSAC Conf., IEEE, Catalog No. 80CH1607-1 (October 17–31, 1980), pp. 19–23.
9. N. Wirth, *Systematic Programming*, Englewood Cliffs, NJ: Prentice Hall, 1973.
10. E. Yourdon and L. Constantine, *Structured Design*, Second Edition, New York: Yourdon Press, 1978.
11. M. A. Jackson, *Principles of Program Design*, New York: Academic Press, 1975.
12. E. W. Dijkstra, "Co-operating Sequential Processes," in *Programming Languages*, F. Genuys, Ed., New York: Academic Press, 1968, pp. 43–112.
13. D. L. Parnas and K. Heninger, "Implementing Processes in HAS," in *Software Engineering Principles*, Naval Research Laboratory, Washington, D. C. 20375, Document HAS.9, 1978.
14. J. C. Mathes and D. W. Stevenson, *Designing Technical Reports*, Indianapolis: Bobbs-Merrill, 1976.
15. D. L. Parnas, "On the Design and Development of Program Families," IEEE Trans. Software Engineering, *SE-2*, (March, 1976), pp. 1–9.
16. S. B. Sheppard, E. Kruesi, and B. Curtis, "The Effects of Symbology and Spatial Arrangement on the Comprehension of Software Specifications," Proc. Fifth Int. Conf. Software Engineering, ACM Order No. 592810 (1981), pp. 207–14.